1 Objectives

The lack of mutable variables gives us the ability to perform many analyses using mathematics. In this lecture we talk about *equational reasoning* and references, and see techniques for limiting the scope of the state to improve the reliability of your code.

- Be able to explain equational reasoning and give an example.
- Know the syntax of references in OCaml.
- Know the tradeoffs between imperative and functional features.
- Know the constructions to define a function with local state.
- Be able to state the benefits of local state and give an example.
- Be able to use tuples to allow multiple functions access to the same state.

2 Examples

Counter, version 1.

```
# let ct = ref 0;;
val ct : int ref = {contents=0}
# let counter () =
    ct := !ct + 1;
    !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

Counter, version 2.

```
# let counter =
    let ct = ref 0 in
    fun () -> ct := !ct + 1; !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

Why is version 2 okay but version 1 is bad?

3 Problems

Try the following problems. In a few minutes the instructor will go over the solutions. Feel free to work with the person next to you!

1. Supposing you wanted a counter that did *not* use references, how would you go about writing it?

2. The random number function generator does not have a way to reset the state. We would also like to be able to ask "what was the last random number generated" without changing the seed. Write a (group of) functions to do this.

3. Suppose we want a more generic way to represent counters—in fact, suppose you want *several* counters in your program. You could just repeat the code several times, but there are serious flaws to that approach. What are they? How might you go about fixing them?